EECS150: Spring 2010 Project Checkpoint 4, Line Engine

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

Revision C

1 Time Table

ASSIGNED	Saturday, April 17^{th}
DUE	Week 15: April $27^{th} - 29^{th}$, 10 minutes after your lab section starts

2 Overview

Now that the basic infrastructure for the video subsystem is complete, we can display an entire frame of video on the screen. Still, however interesting the frame is, it is just a single frame (a static image!). In order to make the video subsystem useful, we must have a means of changing the image that is to be displayed.

What immediately comes to mind is to just change the contents of the frame buffer through the MIPS150 processor. This definitely works: for example, to clear the entire screen, we could perform 480,000 **sw** operations (one for each pixel on the screen) and write the same color into all locations. As it turns out, however, this approach has several shortcomings. First, it is memory intensive: we have to allocate space for the **sw** instructions.[1] Second, it wastes CPU time: the MIPS150 processor actually has to perform all of the **sw** operations! Regardless of how we organize our code, given that the MIPS150 can only perform 1 **sw** per cycle, it will always take *at least* 480,000 CPU cycles to redraw an entire frame. It will take less cycles if we only want to change a part of the frame, but it will consume CPU cycles regardless.

In order to free up CPU time and resources when performing common graphics routines, we will build a **Line Drawing Engine** (or "Vector Accelerator Engine") that works alongside the processor and frame buffer. The line engine is the deliverable in checkpoint 4. The job of the line engine is to quickly draw a line from one set of x, y coordinates to another. This block will not only speed up the line drawing process but also allow for the CPU to be doing other work while lines are being drawn.

We will be using the **Bresenham Line Drawing Algorithm** to to implement the the line engine. The algorithm is explained later in this document using C. Your job will be to create a hardware implementation for the given software routine.

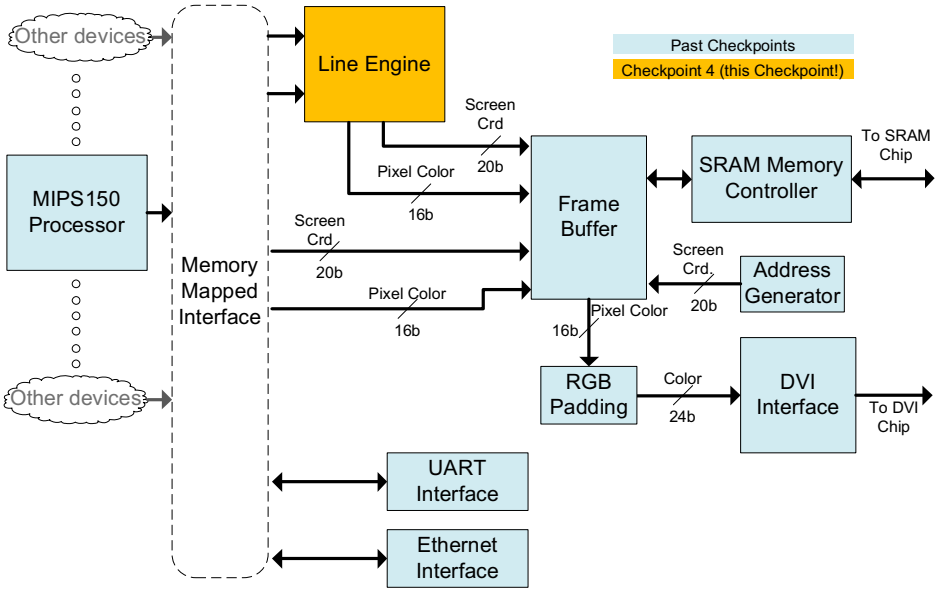
3 Checkpoint Components

Your project, factoring in work done for checkpoint 3, now looks like what is shown in Figure 1.

Your goal in this checkpoint will be to implement the line engine in hardware. As mentioned in Section 2, we have provided the software routine, that represents what your hardware implementation will do, in Program 1.

[1] Depending on how we code the "clear screen" operation, it would take more or less space in instruction memory to store all of the instructions necessary. For this example, we assume that all of the **sw** operations are in memory back-to-back to optimize performance.

Figure 1 Your project circa checkpoint 4.



The implementation shown in Program 1 works for drawing lines of any slope and for lines in any quadrant of the 2D plane. Looking at the software implementation, think about how the MIPS150 processor would perform the line drawing algorithm if it had to in software. How long would drawing an entire line take?

Given all of this, there is still a lingering issue: how the line engine will be told to start drawing a line (and which line to draw). As with the other devices talking to MIPS150, the line engine communicates over the memory mapped interface. Specifically, it occupies three addresses as shown in Table 1.

In Table 1, **Ready** is a single bit control signal that the MIPS150 CPU will poll to see if the line engine is ready to receive new $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{y}_0$, $\mathbf{y}_1$, and **color** values. The coordinate values are all 10-bit values and **color** is 16-bits.

The operation of the line engine is as follows:

1. On system reset, the line engine will set the **Ready** bit.
2. When the CPU is ready to draw a line, it will poll the **Ready** bit.
3. Since the **Ready** bit was high, the CPU will write $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{y}_0$, $\mathbf{y}_1$, and **color** into their appropriate locations.
   (a) Writing to **0x8040_004{0,4,8,c}** and **color** (corresponding to Non-trigger registers for coordinates and the color register, respectively) will simply tell the line engine about that value.
   (b) Writing to **0x8040_005{0,4,8,c}** (corresponding to the Trigger registers for coordinates) will tell the line engine about that value ***and* tell the line engine to start drawing the line.**
4. The line engine will clear **Ready** while it is drawing the line (the CPU must be sure to not modify any of the line engine's memory map locations during this time).
5. When the line engine is finished, it will set **Ready** again and the CPU will be able to write more coordinates.

This scheme enables some optimizations on the software side. If you want to draw multiple lines that share everything but a single x or y value, simply write to the corresponding trigger register multiple times in a row without writing to other registers. (Obviously, many other combinations of trigger/non-trigger registers are possible.) Try to come up with some interesting algorithms for drawing unique pictures!

Program 1 Bresenham Line Drawing Algorithm in C.

```
#define SWAP(x, y) (x ^= y ^= x ^= y)
#define ABS(x) (((x)<0) ? -(x) : (x))

void line(int x0, int y0, int x1, int y1) {
    char steep = (ABS(y1 - y0) > ABS(x1 - x0)) ? 1 : 0;
    if (steep) {
        SWAP(x0, y0);
        SWAP(x1, y1);
    }
    if (x0 > x1) {
        SWAP(x0, x1);
        SWAP(y0, y1);
    }
    int deltax = x1 - x0;
    int deltay = ABS(y1 - y0);
    int error = deltax / 2;
    int ystep;
    int y = y0
    int x;
    ystep = (y0 < y1) ? 1 : -1;
    for (x = x0; x <= x1; x++) {
        if (steep)
            plot(y,x);
        else
            plot(x,y);
        error = error - deltay;
        if (error < 0) {
            y += ystep;
            error += deltax;
        }
    }
}
```

Table 1 Map of the line engine address space.

	Addresses	Read/Write	Composed of ...
Control register	0x8040_0064	R	Ready
Color register	0x8040_0060	W	color
Trigger registers	0x8040_005c	W	y_1
	0x8040_0058	W	x_1
	0x8040_0054	W	y_0
	0x8040_0050	W	x_0
Non-trigger registers	0x8040_004c	W	y_1
	0x8040_0048	W	x_1
	0x8040_0044	W	y_0
	0x8040_0040	W	x_0

4 Architectural Concerns

Your goal after implementing the line drawing algorithm shown in Program 1 is to write one new pixel to the frame buffer every cycle. Contrast this performance with that which you would be able to obtain from the MIPS150 processor running the algorithm in pure software. Know that in order to meet timing, you may have to pipeline your implementation.

One of the last things to consider is the fact that the Line Engine is not the only thing trying to access the SRAM. There will be times when the Line Engine is attempting to write during a cycle when the SRAM arbiter is not able to accept a write from the Line Engine. To support flexibility in memory arbitration, your Line Engine must also support the ability to stall. It should accept a signal, `stall`, that, whenever high, halts the operation of your Line Engine's pipeline. Line coordinates that are not able to be written must not be lost. The Line Engine should commit writes whenever it can.

Rev.	Name	Date	Description
C	Kyle Wecker	4/17/2010	Modified to be consistent with the Spring 2010 project specification.
B	Chris Fletcher	4/21/2009	Clarified how the line engine starts an operation (added information about the difference between writing to address **0x8040_0040** and address **0x8040_0044**.
A	Chris Fletcher Ilia Lebedev	3/26/2009	Wrote new Document